Check for updates

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2020;**30**:e1731.
Published online in Wiley Online Library (wileyonlinelibrary.com). https://doi.org/10.1002/stvr.1731

SPECIAL ISSUE PAPER

Fully Automated HTML and JavaScript Rewriting for Constructing a Self-healing Web Proxy

Thomas Durieux[1,*,†], Youssef Hamadi[2] and Martin Monperrus[3]

[1]*INESC-ID and IST, University of Lisbon, Lisbon, Portugal*
[2]*Uber Elevate Research, Paris, France*
[3]*KTH Royal Institute of Technology, Stockholm, Sweden*

SUMMARY

Over the last few years, the complexity of web applications has increased to provide more dynamic web applications to users. The drawback of this complexity is the growing number of errors in the front-end applications. In this paper, we present an approach to provide self-healing for the web. We implemented this approach in two different tools: (i) BikiniProxy, an HTTP repair proxy, and (ii) BugBlock, a browser extension. They use five self-healing strategies to rewrite the buggy HTML and JavaScript code to handle errors in web pages. We evaluate BikiniProxy and BugBlock with a new benchmark of 555 reproducible JavaScript errors of which 31.76% can be automatically self-healed by BikiniProxy and 15.67% by BugBlock. © 2020 John Wiley & Sons, Ltd.

Received 17 March 2019; Revised 30 December 2019; Accepted 2 January 2020

KEY WORDS: self-healing; bugs; JavaScript; proxy; chrome extension

1. INTRODUCTION

According to the literature[1], at least 76% of all websites on the Internet use JavaScript. The JavaScript code used in today's web pages is essential: it is used for social media interaction, dynamic user interface, usage monitoring, advertisement, content recommendation, fingerprinting and so forth, all of this being entirely part of the "web experience." For example, when a user browses the website cnn.com, he or she is loading more than 125 JavaScript files, which represent a total of 2.8 mb of code.

The drawback of this complexity is the growing number of errors in web pages. For instance, a common JavaScript error is due to uninitialized errors, resulting in an error message such as `cannot read property X of null`. Ocariza et al. [2] have performed a systematic study showing that the majority of the most visited websites contain JavaScript errors.

In this paper, we propose a novel technique to provide self-healing for the web. It is along the line of previous work on self-healing software [3, 4], also called failure-oblivious computing [5, 6], automated recovery and remediation [7]. The majority of the self-healing literature focuses on the C/Unix runtime. On the contrary, we are interested in the JavaScript/browser runtime, which is arguably much different. Indeed, the topic of self-healing for the web is a very little researched area [8, 9].

Our novel self-healing technique is founded on two insights. Our first key insight is that proxying the source code before it is executed can be used for providing self-healing capabilities to an appli-

*Correspondence to: Thomas Durieux, INESC-ID and IST, University of Lisbon, Portugal.
†E-mail: thomas@durieux.me

© 2020 John Wiley & Sons, Ltd.

cation. In this paper, we focus on two types of proxy: (i) an HTTP proxy between the browser and the web server and (ii) a browser extension that modifies the request inside the browser. Our second key insight is that the most common JavaScript errors can be fixed by an automated rewriting of HTML or JavaScript code.

In this paper, we present two novel self-healing tools: a proxy for the web called BikiniProxy and a browser extension called BugBlock. The two tools contain five self-healing strategies that are specifically designed for JavaScript errors. Those strategies are based on rewriting, defined as an automated modification of the code. BikiniProxy and BugBlock automatically modify the Document Object Model (DOM) of HTML pages or automatically transforms JavaScript abstract syntax trees (AST).

Our approach does not make any assumption on the architecture or libraries of web applications. First, proxy servers are used in most web architectures, and browser extensions are commonly used to change the browser behaviour, for example, to block ads. Second, our approach does not require a single change to existing web pages and applications. As such, it is highly applicable.

We evaluate our approach as follows. First, we set up a crawler to randomly browse the Internet; for each browsed page, it logs the JavaScript errors, if any, occurring during the loading of the page content. Second, we observe how BikiniProxy and BugBlock heal those errors by collecting and comparing traces. Over eight full days, our crawler has visited 96 174 web pages and identified 4282 web pages with errors. We observed that 3727 errors were either transient (due to asynchronicity [2]) or fixed by the developer after crawling. Eventually, we evaluated BikiniProxy and BugBlock on 555 web pages with errors, representing a random sample of real field errors. BikiniProxy is able to make all errors disappear in 176/555 of the cases, that is, 31.76%. BugBlock is able to make all errors disappear in 87/555 of the cases, that is, 15.67%. In the best cases, the self-healing provides the user with new features or new content. We provide a detailed qualitative and quantitative analysis of the main categories of self-healing outcome. To sum up, BikiniProxy and BugBlock are novel fully automated self-healing techniques that are designed for the web, evaluated on 555 real JavaScript errors and based on original self-healing rewriting strategies for HTML and JavaScript.

Our contributions are as follows:

- A novel self-healing approach for today's web, implemented into two independent tools: a self-healing proxy (BikiniProxy) and a self-healing browser extension (BugBlock).
- Five self-healing strategies for the web, specifically designed to automatically recover from the most frequent JavaScript errors happening in the browser.
- A benchmark of 555 real web pages with JavaScript errors. Special care has been taken so that all errors are fully reproducible for future experiments in this research area.
- An evaluation of BikiniProxy and BugBlock over 555 JavaScript errors from our benchmark. It shows that BikiniProxy makes all errors disappear for 31.76% of web pages with errors and BugBlock handles all the errors in 15.67% of the web pages from our benchmark. This quantitative evaluation is complemented by a qualitative analysis of BikiniProxy's and BugBlock's effectiveness.
- The implementations of BikiniProxy and BugBlock are open-source and publicly available for future research [10].

This paper is an extended version of the study of Durieux et al. [11]. In the conference paper, we presented only BikiniProxy and its evaluation. In this extension, we devise and evaluate the corresponding browser extension BugBlock. Both tools have the same purpose and target the same failures, but they target different use cases and execution environments. Having both shows that our underlying self-healing concept is generic and widely applicable.

The remainder of this paper is organized as follows. section 2 explains the background of BikiniProxy. section 3 details our approach for introducing self-healing web application and two implementations: BikiniProxy and BugBlock. section 4 details the evaluation. section 5 discusses the security and applicability aspects of the approaches. section 6 details the threats to validity of the contribution. section 7 presents the related works, and section 8 concludes.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2. BACKGROUND

2.1. The complexity of today's web

A web page today is a complex computational object. A modern web page executes code and depends on many different resources. Those resources range from CSS styles of thousands of lines, external fonts, media objects and, last but not least, JavaScript code.

For example, when a user browses the website cnn.com, he or she is loading more than 400 resources, and 125 of them are JavaScript, which represents a total of 2.8 mb of code. Anecdotally, back in 2010, the same web page cnn.com contained 890 kb of JavaScript code [2] (68% less code!).

Today's web page JavaScript is essential: it is used for social media interaction, usage monitoring, advertisement, content recommendation, fingerprinting and so forth, all of this being entirely part of the "web experience." Consequently, 76% of all websites on the Internet use JavaScript [1]. To this extent, a web page today is a program and, as such, suffers from errors.

2.2. JavaScript errors

Web pages and applications load and execute a lot of JavaScript code [12]. This code can be buggy; in fact, the top 100 of the most visited websites contains JavaScript errors [2]. One kind of JavaScript error is an uncaught exception, which is similar to uncaught exceptions in modern runtimes (Java, C#, Python). While the JavaScript community uses the term "error," the research community use "failure" in this case. Those errors are thrown during execution if the browser state is invalid as when accessing a property on a null element (a null dereference).

If an error is not caught by the developer, the execution of the current script is stopped. In JavaScript, there is a different "execution scope" for each loaded scripts (i.e. for each HTML script tag) and for each asynchronous call. Consequently, contrary to classical sequential execution, in a browser, only the current execution scope is stopped, and the main thread continues running. This means that one can observe several uncaught exceptions for a single page. Therefore, depending on where the error is happening, the error is perceived differently by the end-user. If the error happens in the main execution scope, the failure will prevent the execution of all the code that is after the crash. This will have an important impact on the user experience. However, if the error happens in an execution scope that is not meant to provide a feature for the user, the user will not perceive it, and it will give the impression that the browser tolerates the failure. But, in this case, there can still be an observable impact on the system, for example, if the error happens in the logging module.

The uncaught errors are logged in the browser console that is accessible with the developer tool. Most browsers provide an API to access all the errors that are logged in the browser console. This means that it is relatively easy to monitor JavaScript errors in web applications.

2.3. Web proxies

A web proxy is an intermediate component between a client browser and a web server. In essence, a proxy transmits the client request to the server, receives the response from the server and forwards it to the client browser. On the web, proxies are massively used for different purposes.

> A `Network Proxy` is used to expose a service that is not directly accessible because of network restrictions [13]. The proxy, in this case, is a bridge between two networks, and its only task is to redirect requests and responses. For example, a popular network proxy is Nginx. It is used to expose websites on port 80 that are indeed served on other ports >1024. This avoids granting root access to web applications.
>
> A `Cache Proxy` is a proxy that is used to cache some resources directly in the proxy in order to improve the serving time to an external resource [14]. The cache proxy stores the response of the server locally, and if a request is made for the same resource, the local version is directly sent to the client without sending the request to the server. A widely used cache proxy is, for example, a content-delivery network that provides optimized access to static resources on the Internet.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

A `Security Proxy` is used to verify whether a client browser is legitimate to access a server [15]. This type of proxy can be used, for example, to protect a server against denial-of-service attacks.

A `Load-balancer Proxy` is used on popular applications to distribute the load of users on different backend servers [16]. A load balancer can be as simple as a round-robin but can also be more sophisticated. For instance, a load balancer can try to find the least loaded server available in the pool.

2.4. Browser extensions

Browser extensions are used by end-users to extend the functionalities of their browsers. It is used for changing the user interface of the browser, for blocking ads and for increasing the privacy.

The browser extensions are HTML/JavaScript micro-applications that have more permissions than the traditional browser JavaScript. Those applications work at three different levels in the browser. The first level is called the "background." It means that the code in the extension constantly runs and is generally used for the core behaviour of the extension and the state storage. It is also the only part of the extension that can register to specific events, such as a new request event and a new tab event. The second level is the "action." It is used for providing information to the user. It is generally presented as an icon on the right of the URL bar. This level has access to the state of the first level. The third level is the "injected script." This level is used to inject specific JavaScript behaviour on the page. It can be used, for example, by a password manager to inject the login and password in a login form.

With those three levels, browser extensions have the possibility to drastically change the behaviour of web pages and interact with the users. One way to modify the behaviour of web pages is to block, redirect and inject scripts in web pages. For example, ads blockers block the HTTP requests that match specific patterns.

3. FULLY AUTOMATED SELF-HEALING APPROACH AND TOOLS

We now present our novel approach to fully automate self-healing of HTML and JavaScript in production. This approach is composed of three main parts: firstly, the failure-oblivious computing central component (subsection 3.1), secondly, the self-healing strategies that are used to inject failure-oblivious properties to an application (subsection 3.2) and, finally, the two implementations of the approach BikiniProxy, an HTTP proxy (subsection 3.3), BugBlock, a browser extension (subsubsection 3.4.1).

Those two different implementations show the power of our self-healing concept in different environments. The first one is more powerful but less practical (BikiniProxy), and the second one is less powerful but more practical (BugBlock).

3.1. Failure-oblivious computing

Our work is founded on the failure-oblivious computing principle [5].

In the failure-oblivious computing principle, it is possible to perform speculative execution after the failure point instead of crashing.

The goal of failure-oblivious computing is to provide a degraded behaviour instead of no behaviour at all when the application is crashing. It is not a long-term solution, but it increases availability. It can be compared with a temporary hotfix to mitigate a problem before it is permanently fixed. Failure-oblivious computing is desirable when availability must be maximized while having limited resources for engineering expensive highly reliable and available software. There is indeed an engineering trade-off between cost and reliability [17]. Failure-oblivious computing fits in the general context of self-healing software [3, 4], that is, software with seatbelts and airbags [18]. Self-healing is also known as runtime repair [19] or automated recovery [20, 21].

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.2. Self-healing strategies

This section presents the five self-healing strategies that we designed and implemented to mitigate JavaScript errors. We designed those strategy to target the most frequent JavaScript errors happening on the web (see subsection 4.2 for how we identify those most common errors). Those fives strategies are implemented in two tools: in BikiniProxy and in BugBlock.

We designed the following:

1. HTTP/HTTPS Redirector that changes HTTP URLs to HTTPS URLs (subsubsection 3.2.1).
2. HTML Element Creator that creates missing HTML elements (subsubsection 3.2.2).
3. Library Injector injects missing libraries in the page (subsubsection 3.2.3).
4. Line Skipper wraps a statement with an if to prevent invalid object access (subsubsection 3.2.4).
5. Object Creator initializes a variable with an empty object to prevent further null dereferences (subsubsection 3.2.5).

3.2.1. HTTP/HTTPS Redirector. The modern browsers have the policy to block unsecured content (i.e. HTTP resources) in secured web pages (HTTPS). For example, https://foo.com cannot load http://foo.com/f.js (no https). The rationale is that the unsecured requests can be easily intercepted and modified to inject scripts that will steal information from the secure page, for example, your banking information.

As of 2019, we still are in a period of transition where both HTTP and HTTPS web pages exist and some websites provide access to their content with both HTTP and HTTPS protocols. Consequently, it happens that the developers forget to change some URL in their HTTPS version, and those resources are blocked, resulting in incomplete web pages or JavaScript errors.

The self-healing strategy of HTTP/HTTPS Redirector is to change all the HTTP URL by HTTPS URL in HTTPS pages. By doing this, all resources are loaded, and the unwanted behaviour due to blocked resources is fixed.

3.2.2. HTML Element Creator. As shown by Ocariza et al. [12], most of the JavaScript errors are related to the DOM. This is especially true when the developers try to dynamically change the content of a specific HTML element using getElmentById, like:

```
document.getElmentById("elementID").innerText = "Dynamic
content";
```

Because the HTML and the JavaScript are provided in different files, it is not rare that an HTML element with an ID is removed or changed without changing the associated JavaScript code. For example, if DOM element "`elementID`" is removed, `document.getElmentById(...)` returns `null`, and the execution results on a null dereference when the property `innerText` is set.

When an error happens, BikiniProxy and BugBlock determine if the error is related to the access of a missing element in the DOM: it does so by looking at the JavaScript code at the failure point. If it is the case, "HTML Element Creator" extracts the query that the JavaScript used to access the element and create an empty and invisible HTML element in the DOM. The JavaScript code then runs without error, and the execution continues without affecting the client browser.

3.2.3. Library Injector. In JavaScript, it is a common practice to rely on external libraries to facilitate the development of the web applications. Some of these libraries are extremely popular and are used by millions of users every day, like jQuery or AngularJS. Sometimes, these libraries are not correctly loaded into web pages, and this produces a very characteristic error: for example, `jQuery is not defined`. In those cases, we can parse the error message and determine which library is missing.

To do so, we realize an initial offline training phase, missing libraries are simulated on a test website, and reference errors are collected for the top 10 libraries presented in the literature [1]. Based on these reference errors, error parsing rules have been manually written to determine which library is missing. When Library Injector detects that a web page contains an error related to a

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

missing library, it injects the related library on the web page. The rewritten page contains the missing library, and the web page can be completely loaded.

Algorithm 1 Algorithm to rewrite JavaScript code with "Line Skipper" strategy

Require: E: error
Require: R: resource
1: (line, column, resource_url) = extractFailurePoint(E, R.body)
2: **if** resource_url is R.url **then**
3: ast = getAST(R.body)
4: elem = getElementFromAst(ast, line, column)
5: wrapElemWithIf(elem)
6: R.body = writeAST(ast)
7: **end if**

3.2.4. Line Skipper. The errors `XXX is not defined` and `XXX is not a function` in JavaScript are errors that are related to invalid access to a variable or a property. `XXX is not defined` is triggered when an identifier (name of variable/function) is used but was never defined in the current scope. For example, if we call `if(m){}` without defining `m`, the execution ends with the error `'m' is not defined`.

The second error, `XXX is not a function`, is triggered when a variable that is not a function is called. For example, the code `var func = null; func()` will trigger the error `func is not a function`.

To avoid these errors, Line Skipper wraps the statement that contains the invalid code with an if that verifies that the element is correctly defined for the first error, `if (typeof m != 'undefined' && m) {if(m){}}`. And to verify that a variable contains a reference to function, the rewriter changes the JavaScript code as follows `if (typeof func === 'function') {func()}`.

Algorithm 1 presents the algorithm of Line Skipper. First, Line Skipper extracts the line, column and URL of the resource from the failure point of the error. Line Skipper verifies that the current request is the resource that contains the error. Then, Line Skipper extracts the AST from the JavaScript code and looks for the element that is not defined. Finally, the AST is transformed back to a textual form to be sent back to the client.

3.2.5. Object Creator. The Object Creator strategy is created to handle null dereferences (e.g. `NullPointerException` in Java), which is one of the most frequent error type [22] in Java and also the most frequent failure in JavaScript [12]. The typical strategy to handle those errors is to initialize the null variable or replace the null expression by another non-null variable or expression. Because JavaScript is an untyped language, all null dereferences can be handled by initializing the null variable/expression with a generic empty object `var obj = ;`. For example, the code `var m = null; m.test = ";` will trigger the error `Cannot set property test of null` and can be handled by adding `if (m == null) {m = {};}` before setting `m.test`.

3.3. BikiniProxy

We now are presenting BikiniProxy, the first of the two implementations of our approach. The intuition behind BikiniProxy is that a proxy between web applications and the end-users could provide the required monitoring and intercession interface for automatic error handling. This is the concept of "self-healing proxy" that we explore in this paper. BikiniProxy is composed of three main parts that are presented in Figure 1.

1. Proxy (subsubsection 3.3.1) is a stateless HTTP proxy that intercepts the HTML and JavaScript requests between the browser (also called "client" in this paper) and the web server.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

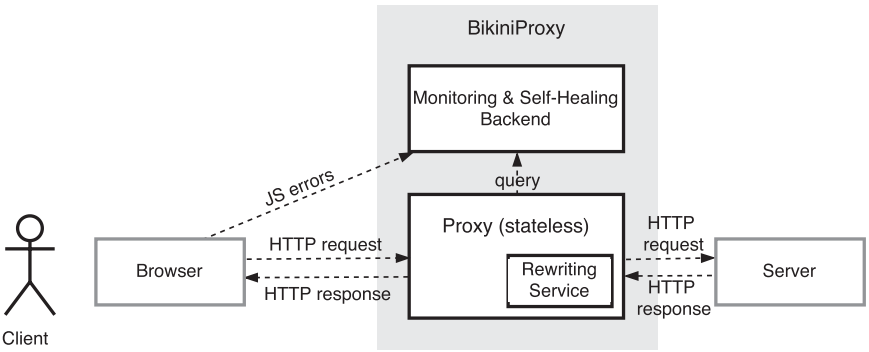


Figure 1. The architecture of BikiniProxy. The key idea is that all requests are proxied by "BikiniProxy." Then, if an error is detected, a self-healing strategy based on HTML and/or JavaScript rewriting is automatically applied.

2. The Rewriting Service (subsubsection 3.3.2) that contains the self-healing strategies to handle JavaScript errors.
3. The Monitoring and Self-healing Backend (subsubsection 3.3.3) stores information about the known errors that have happened and the success statistic of each self-healing strategies for each error.

Let us start with a concrete example. Bob, a user of the website http://foo.com, browses the page `gallery.html` and uses BikiniProxy to improve his web experience. Because BikiniProxy is a proxy, when Bob opens `gallery.html`, the request goes through the proxy. When the request is made, BikiniProxy queries the backend to know whether another, say Alice, has experienced errors on `gallery.html`. Indeed, Alice's browser got a `jQuery is not defined` error 2 days before. The backend sends this error to the proxy, which consequently launches the Rewriting Service to handle the error. For `gallery.html`, the rewriting is HTML-based and consists of injecting the library jQuery in the HTML response. BikiniProxy also injects its error monitoring framework before sending the rewritten response to Bob's browser. The rewritten page is executed by Bob's browser; BikiniProxy's monitoring tells the proxy that Alice's error does not appear anymore, meaning that the self-healing strategy handled it.

Algorithm 2 shows the complete workflow of BikiniProxy. BikiniProxy receives the HTTP request from the browser (Line item 1). Then, it redirects the request to the web server (Line item 2) like any proxy. For each HTML response, BikiniProxy injects a framework (Line item 4) to monitor the JavaScript errors in the client browser. When an error happens on the client browser, it is sent to the BikiniProxy's backend for being saved in a database.

BikiniProxy queries the Monitoring & Self-healing backend to know which JavaScript resource has thrown an error in the past: for each HTML and JavaScript resource, BikiniProxy queries the backend service with the URL of the requested resource to list all the known errors (Line item 6). If there is at least one known error, BikiniProxy triggers the Rewriting Service to apply the self-healing strategies to the requested resource (Line item 10). Then, the response is sent to the client (Line item 15) with a unique ID to monitor the effectiveness of the applied self-healing strategy.

A web **resource** is a content on which a web page is dependent. For instance, an image or a JavaScript script is a web resource. In this paper, a web resource is defined by (i) a URL to address the resource, (ii) its content (text or binary content) and (iii) the HTTP headers that are used to serve the resource. The resource can be used as an attribute of an HTML tag (<script>, <img>, <link >, <iframe> etc.) or used as an AJAX content.[‡]

[‡]AJAX means requested programmatically in JavaScript code.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 2 The main BikiniProxy algorithm

Require: B: the client browser
Require: W: the web server
Require: R: the rewriting services
Require: D: BikiniProxy backend
while new HTTP $request$ from B **do**
  $response \leftarrow \mathrm{W}(request)$
  if request is html page **then**
    $response \leftarrow inject\_bikiniproxy\_code(response)$
  end if
  $errors \leftarrow D.previous\_errors\_from(request_{url})$
  if errors is not empty **then**
    for r in R **do**
      if $isToApply(r, errors, request, response)$ **then**
        $response \leftarrow r.rewrite(response, request, errors)$
        $response \leftarrow response + uuid$
      end if
    end for
  end if
  $send(response)$
end while

3.3.1. The Proxy. A proxy intercepts the HTML code and the JavaScript code that is sent by the web server to the client browser. By intercepting this content, the proxy can modify the source code of the website and therefore change the behaviour of the web application. One well-known example of such a change is to minimize the HTML and JavaScript code to increase the download speed.

In BikiniProxy, the proxy automatically changes the JavaScript code of the web application to handle known errors. BikiniProxy is configured with what we call "self-healing strategies." A self-healing strategy is a way to recover from a certain class of errors automatically. The strategies are presented in subsection 3.2 and how they are applied is presented in subsubsection 3.3.2.

3.3.2. Rewriting Service. The role of the Rewriting Service is to rewrite the content of the JavaScript and HTML resources in order to (i) monitor the JavaScript errors that happen in the field and (ii) change the behaviour when a JavaScript resource has been involved in an error in the past. In this paper, a "known error" is an error that has been thrown in the browser of a previous client, which has been detected by the monitoring feature of BikiniProxy and has been saved in the Monitoring & Self-healing Backend (subsubsection 3.3.3).

We design five self-healing strategies that target the most frequent JavaScript errors that we observe when we craw the Internet. Those strategies are presented in subsection 3.2. In addition, the Rewriting Service is plugin-based, and it can be easily extended with new self-healing strategies to follow the fast evolution of the web environment.

3.3.3. Monitoring and Self-healing Backend. The Monitoring and Self-healing Backend fulfills three tasks. The first task is to receive and store all the JavaScript errors happening on client browsers. The Backend provides an API for BikiniProxy to query if a specific resource (URL) contains known errors.

The second task of the backend is to monitor the effectiveness of the different self-healing strategies. Each time that the section of the JavaScript code rewritten by one of the five self-healing strategies is executed, an event is sent to the BikiniProxy backend to keep track of the activation of the different strategies. Based on the number of activation and the number of errors per page, we can estimate the relative effectiveness of the self-healing strategies.

The third task is to provide a layer of communication with the developers about the monitored errors and the effectiveness of all self-healing strategies. For example, the following message can

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

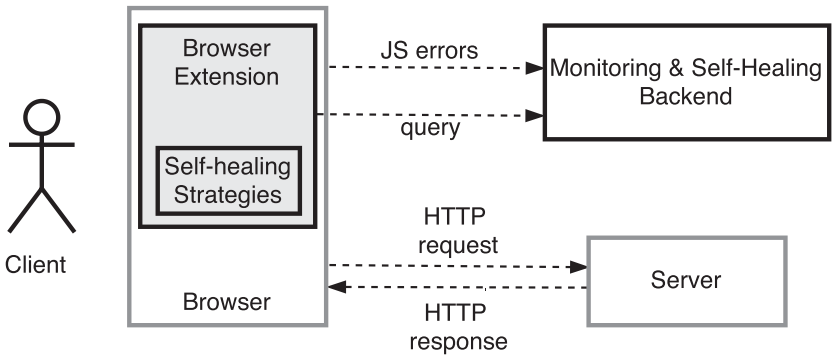


Figure 2. The architecture of BugBlock. The key idea is that the browser extension intercepts the HTTP requests of the server and applies a self-healing strategy when an error is known in the request directly in the browser.

be given to the developer: "The strategy `Library Injector` has injected jQuery 22 times in the page `gallery.html` to handle the error `jQuery is not defined`." This is valuable information to assist the developers in designing a permanent fix. The backend also provides a visual interface that lists all the errors that the end-users face during the browsing of the web page.

3.4. BugBlock

We now present the components of the second implementation of our approach. BugBlock is the second implementation of our approach that aims to provide self-healing abilities to web applications. BugBlock is like an ad-blocker, but instead of blocking the advertisement, it blocks the JavaScript errors. This implementation is independent of BikiniProxy.

Figure 2 presents the architecture of BugBlock. It is composed of two main parts.

1. BugBlock (subsubsection 3.4.1) is a browser extension that intercepts the HTML and JavaScript requests between the browser and the page rendering. The extension is also responsible for applying the self-healing strategies that handle JavaScript errors.
2. The Monitoring Backend (subsubsection 3.4.3) stores information about the known errors that have happened and the success statistics of each self-healing strategy for each error. This part can be share with the Monitoring and Self-healing Backend of BikiniProxy.

3.4.1. The Extension. The extension part of BugBlock contains all the logic required to apply the self-healing strategies to automatically handle JavaScript errors directly inside the browser.

Algorithm 3 shows the workflow of BugBlock. BugBlock listens to all web requests for each tab in the browser (Line 1). For each main request of a tab (the request that corresponds to the URL of the bab), BugBlock requests the backend to know which JavaScript resource has thrown an error in the past (Line 3), and it injects a monitoring script in the tab (Line 4). When an error happens on the client browser, it is sent to the BikiniProxy's backend for being saved in a database.

For all the other requests, BugBlock checks if they triggered errors in the past (Line 6). If it is the case, it requests the response of the request to the Web Server (Line 7). Then, it applies the self-healing strategies on the requested content (Line 10). BugBlock redirects the request to data format URL (Line 13) that allows to send content to the browser in the base64 format, for example, `data:text/javascript,<base64>`.

3.4.2. Interface. BugBlock is a browser extension that has a similar user interface to ad-blockers. It is directly integrated into the browser, displays its logo next to the URL bar and provides feedback to the users about what has been detected and what code transformations have been made and executed.

Figure 3 presents a screen capture of the user interface. It provides the list of known errors for the page, the list of errors that are currently faced and the list of the self-healing strategies that have

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 3 The main BugBlock algorithm

Require: T: a browser tab
Require: W: the web server
Require: R: self-healing strategies
Require: D: BikiniProxy backend

```
1:  while new request to T do
2:      if request is main request then
3:          errors ← D.previous_errors_from(request_url)
4:          inject_bikiniproxy_code in T
5:      end if
6:      if request triggers an error from errors then
7:          response ← W(request)
8:          for r in R do
9:              if isToApply(r, errors, request, response) then
10:                 response ← r.rewrite(response, request, errors)
11:             end if
12:         end for
13:         redirect(response)
14:     end if
15: end while
```

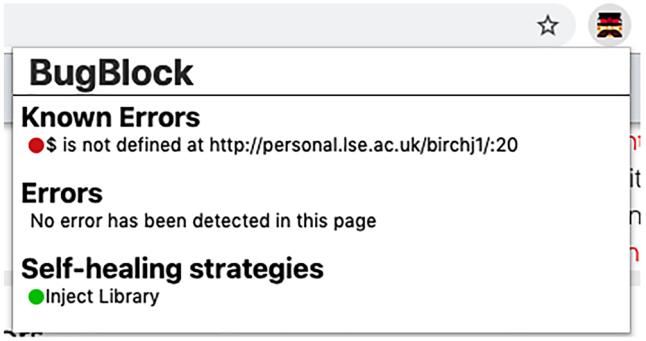


Figure 3. The user interface of BugBlock. It shows the known errors, the errors that are currently detected and the self-healing strategies that are applied.

been applied to the current page. The interface can be extended to provide further statistics to the users, such as the most frequent errors and the websites that contain the biggest number of errors per visited page. Those statistics can help to educate the user to know what is happening on their web session.

3.4.3. Monitoring and Self-healing Backend. The Monitoring and Self-healing Backend of BugBlock is the same component as the one present in BikiniProxy. If some users use the proxy, while others use the browser extension, having the same component allows our system to share the self-healing knowledge between the two groups of users. In theory, this component could also be directly integrated into the browser extension. This would allow us to have a complete decentralized approach and to increase the privacy because no content will ever be sent to an external server.

3.5. Difference between BikiniProxy and BugBlock

BikiniProxy and BugBlock are two different implementations of the same approach. They aim to provide self-healing ability to web applications. BikiniProxy is an HTTP proxy, and BugBlock is

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table I. Summary of the differences between BikiniProxy and BugBlock.

Characteristics	BikiniProxy	BugBlock
Environment	HTTP proxy	Browser extension
Difficulty of installation	Specialist	End-user
Permission level	Full control of HTTP traffic	Limited by the browser API
Feedback level	Developer feedback via a dashboard	User feedback via the extension and developer feedback via a dashboard
# of self-healing strategies	5	5
Self-healing location	Server-side	Client-side
Update	Handle by the server owner	Handle automatically by the extension store

a browser extension. Both tools intercept HTTP requests to inject self-healing strategies to handle known JavaScript errors.

Despite those similarities, BikiniProxy and BugBlock are different and have different characteristics. Table I highlights their differences. The proxy-based approach of BikiniProxy is the most powerful approach of the two: it allows to modify all the requests that go through the proxy freely. However, it is complex to be set up by end-users and introduces a potential security issue because BikiniProxy behaves as a man in the middle.

On the other hand, the main strength of BugBlock is that it is easy to be set up by end-users. This set up also contributes to limit the performance overhead because all the modifications are made locally. There is no centralized server that needs to handle all the requests from all potential users of the system. However, BugBlock is limited by the API of the browsers. Therefore, it is not possible to modify the scripts that are present in the main HTML page of the website and, consequently, limits the ability of the tool. An additional advantage of BugBlock over BikiniProxy is that it provides a direct feedback to the users, that is, to tell him or her whether a self-healing strategy has been applied (subsubsection 3.4.2). The final advantage of BugBlock is that the implementation is less complex compared with BikiniProxy. This difference in complexity makes BugBlock a more reliable approach compared with BikiniProxy. The technical difference between the two implementations is described in subsection 3.6.

In the evaluation (section 4), we compare the effectiveness of BikiniProxy with BugBlock. The goal is to identify how much is the decrease in effectiveness with the extension-based approach. It is important to analyse this to understand the trade-off between applicability and performance.

3.6. Implementation

In this section, we describe the technical aspects of BikiniProxy and BugBlock. The source code and the usage examples are publicly available on GitHub [10].

3.6.1. Implementation of BikiniProxy. The implementation of BikiniProxy is composed of two main parts: the proxy itself and the code rewriting part.

3.6.1.1. Implementation of the Proxy. As previously explained, BikiniProxy is a technique that intercepts the requests between the clients and the server and modifies them on the fly. This is also known as an HTTP proxy or a man-in-the-middle technique [23] depending on the usage. Because BikiniProxy is designed as a system or browser proxy not as a server proxy, it implies that BikiniProxy will serve content from different servers. This is actually close to a man-in-the-middle technique because it needs to intercept HTTPS requests that are encrypted between the clients and the web servers. In order to decrypt them, we need to perform a man-in-the-middle certificate spoofing. It consists of installing a root certificate in the browser. This root certificate is then used to decrypt all requests between the client and the web server. The proxy then modifies the request, re-encrypts it and finally sends it to the web server. We base the implementation of the proxy on

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

an existing proxy AnyProxy.[§] AnyProxy is a monitoring proxy designed by Alibaba to assist the debugging of their web systems. We modify AnyProxy to be able to modify the requests, and we include the system that allows us to easily add new self-healing strategies.

For the evaluation, the proxy is combined with puppeteer[¶] in order to automate the process.

3.6.1.2 Implementation of Code Rewriting. The second component is responsible for rewriting the source code of the web pages. This poses four main challenges: (i) identify the main request giving the HTML that defines header and body of the page, (ii) regroup all the requests from the same session, (iii) localize the embedded JavaScript scripts in HTML source code and (iv) being fast enough in order not to disturb the user experience.

The two first challenges are related to track the requests and to know which requests need to be rewritten. Indeed, in order to monitor the JavaScript errors, BikiniProxy needs to inject a monitoring script in all the web pages. The only way to inject this script is to identify the main request of the page and to inject the monitoring script inside it. The naive approach of injecting the monitoring script in all HTML requests does not work. Indeed, some HTML requests cannot contain JavaScript code, and therefore, this strategy would introduce additional bugs in the web application under consideration. Our solution is to only inject the script in HTML requests that have a HEAD HTML tag. The drawback of this solution is an increase of required processing, because it requires to analyse all HTML responses.

The second challenge is to link all the JavaScript resources to the main page, for example, the page `foo.com` loads the script `bar.com/jquery.js`, and we need to create a logical link between that resource and the main page `foo.com`. This is required in order to be able to track down which resource contains the bug or which page contains a bug. In order to handle this challenge, the proxy defines a unique ID on each main page. This ID is then used to track the JavaScript resources that are loaded on each page.

The third challenge is to rewrite the JavaScript that is embedded directly in the HTML. This is a problem because we rely on the line number defined in the JavaScript error to rewrite the JavaScript. Because the content of the HTML page can be modified dynamically in JavaScript, the line number of the error can be impacted, which would break the causal relationship between line numbers and errors. We handle this problem by (i) looking at the exact location where the error has been triggered and (ii) verifying that the surrounding lines match the error by looking at the variable names and function name.

The final challenge is about the performance of source code rewriting. Parsing and iterating over the HTML/JavaScript AST is CPU intensive. Therefore, the self-healing strategies have to be optimized to reduce the number of parsed AST and the number of times the AST is traversed. For this, we designed the plugin system for the self-healing strategies in a way that allows to only parse and print once each JavaScript resource. This drastically increases performance.

We use htmlparser2[‖] to parse and iterate the HTML AST and the library babel.js[**] for analysing and transforming the JavaScript abstract syntax tree. The implementation of BikiniProxy is composed of 4378 lines of JavaScript code and 17 dependencies (504.804 lines of code).

3.6.2. Implementation of BugBlock. In this section, we present the prototype implementation of BugBlock. BugBlock does not have challenges 1, 2 and 4 described in the implementation of BugBlock. With a browser extension, we can directly know from which web page a request comes and what the main request is. Therefore, challenges 1 and 2 do not exist. For challenge 4, BugBlock is less impacted by the performance problem because, firstly, it does not need to parse the main request to inject the monitoring script. Secondly, the AST parsing and iterating are directly executed by the client. Therefore, a single server does not need to handle all the load because the load is distributed among all clients.

[§]AnyProxy Github repository: https://github.com/Alibaba/anyproxy.
[¶]Puppeteer GitHub repository: https://github.com/GoogleChrome/puppeteer.
[‖]htmlparser2 Github repository: https://github.com/fb55/htmlparser2.
[**]babel.js GitHub repository: https://github.com/babel/babel.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

However, BugBlock suffers from a different challenge. Browsers do not provide a direct API to modify JavaScript code, which we require to apply the self-healing strategies. The solution that we implemented consists of blocking the request that loads the resource and then create a new HTTP request using the Ajax API that downloads the JavaScript resource as a textual file. Once the file is downloaded, it can be rewritten and injected back to the web page in order to be executed.

The implementation of BugBlock is composed of 1634 lines of JavaScript code and four dependencies (39 866 line of code).

4. EVALUATION

In our evaluation, we answer the following research questions.

RQ1. [Effectiveness] How are BikiniProxy and BugBlock effective at automatically fixing JavaScript errors in production, without any user or developer involvement?

The first research question studies if it is possible to handle field JavaScript errors with our proxy-based approach. We will answer this question by showing how real-world errors have been handled with one of our implemented self-healing strategies.

RQ2. [Outcome] What is the outcome of self-healing strategies with BikiniProxy and BugBlock on the page beyond making the error disappear?

In this research question, we explore what are the possible outcomes of BikiniProxy on buggy web pages. We will answer this research question by presenting real-world case studies of different possible outcomes.

RQ3. [Comparison] Do the different self-healing strategies perform equivalently?

We present to what extent the different self-healing strategies are used: which type of errors are handled by the five self-healing strategies?

4.1. Experimentation protocol

We set up the following experimentation protocol to evaluate BikiniProxy and BugBlock. Our idea is to compare the behaviour of an erroneous web page, against the behaviour of the same, but self-healed page using our tools: BikiniProxy and BugBlock. In order to achieve this goal, we apply the experimentation protocol twice, one for each tool. The comparison is made at the level of "web trace," a concept we introduce in this paper, defined as follows.

A web trace is the loading sequence and rendering result of a web page. A web trace contains (i) the URL of the page, (ii) all the resources (URL, content, see the definition in subsection 3.3), (iii) all the JavaScript errors that are triggered when executing the JavaScript resources and (iv) a screenshot of the page at the end of loading.

Given a benchmark of web pages with JavaScript errors, the following steps are made. The first step is to collect the web trace of each erroneous web page. The second step is to collect the new web trace of each erroneous web page with one of our implementation of proxy-based failure-oblivious approaches (recall that all resources are rewritten by our five self-healing strategies). In addition to the web trace, we also collect data about the self-healing process: the strategies that have been activated, defined by the tuple (initial error, strategy type). The third step is to compare for each web page the original web trace against the self-healed web trace. The goal of the comparison is to identify whether our approach is able to heal the JavaScript errors. For instance, the comparison may yield that all errors have disappeared, that is, a full self-healing.

We apply this protocol twice: once for BikiniProxy and once for BugBlock. At the end of the experimentation, we have for each tool the web trace and the self-healing strategies that have been applied for each bug of the benchmark.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.2. Construction of a benchmark of JavaScript field errors

To evaluate our approaches, we need real-world JavaScript that are reproducible. For each reproducible errors, we want to compare the behaviour of the web page with and without the self-healing approaches. To our knowledge, there is no publicly available benchmark of reproducible JavaScript errors. We create a new benchmark. We call it the DeadClick benchmark. The creation of our benchmark is composed of the following steps:

1. Randomly browses the web to discover web pages on the Internet that have errors (subsubsection 4.2.1).
2. Collect the errors and their execution traces (subsubsection 4.2.2).
3. Ensure that one is able to reproduce the errors in a closed environment (subsubsection 4.2.3).

4.2.1. Web page finder. The first step of the creation of DeadClick is finding web pages that contain errors. In order to have a representative picture of errors on the Internet, we use a random approach. Our methodology is to take randomly two words from the English dictionary and to combine those two words in a Google search request. A fake crawler then opens the first link that Google provides. If an error is detected on this page, the page URL is kept as tentative for the next step. The pros and cons of this methodology are discussed in section 6.

4.2.2. Web Trace Collector. The JavaScript environment is highly dynamic and asynchronous. It means that many errors are transient and as such are not reproducible in the future, even in a very short period of time after their observation.

For identifying reproducible errors, our idea is to collect the web trace of the erroneous page and to try to reproduce the exact same web trace in a controlled environment (subsubsection 4.2.3).

We implement the trace collection using the library puppeteer from Google,[††] which provides an API to control Chrome in a programmatic manner. The big advantage of this library is that it uses the same browser engine as Chrome end-users, meaning that, by construction, DeadClick is only composed of errors that really happen on user browsers.

Because JavaScript is mostly asynchronous, the Web Trace Collector waits for the end of loading where loading is defined as follows: (i) it opens the URL, (ii) it waits for 7 s, in order to load and execute all resources, in particular, JavaScript files, (iii) it scrolls the page to the bottom, in order to trigger additional initialization and JavaScript execution, and (iv) it waits again for 1 s.

During this process, the Web Trace Collector logs (i) all errors that occur in the browser console and (ii) all the requests (including the HTTP headers and the body) made from the browser. When the page is completely loaded, a screenshot of the page is taken, it provides a visual representation of the page. At the end of this process, for each page, the collected data are stored on disk if at least one error has been logged during the page browsing.

4.2.3. Web Page Reproduction. The last step of the benchmark creation consists of verifying that the collected errors can be reproduced. We consider that we succeed to reproduce the behaviour of the web page when the observed errors during reproduction are identical to the ones in the originally collected web trace.

The reproduction of the error is carried out by browsing the erroneous page again, but instead of using the resources from the Internet, the Web Page Reproduction is cut from the Internet and only serves the resources stored on disk. In addition, it denies all the requests that have not been observed during the initial collection of the page.

4.3. Description of DeadClick

Table II gives the main statistics of DeadClick. The Web Page Finder visited a total of 96 174 pages, and 4282 of the pages contains at least one error (4.5%), out of which 555 errors have been successfully reproduced. The final dataset contains errors from 466 different URL domains representing a

[††]Puppeteer repository https://github.com/google/puppeteer.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table II. Descriptive statistics of DeadClick.

Crawling stats	Value
# of visited pages	96 174
# of pages with error	4282 (4.5%)
Benchmarks stats	**Value**
# of pages with reproduced errors	555
# of domains	466
# of average # of resources per page	102.55
# of average scripts per page	35.51
# of minimum errors per page	1
# of average errors per page	1.49
# of maximum errors per page	10
# of average pages size	1.98 mb

Table III. The top 10 error types in DeadClick (left-hand side).

#	Error messages	# of web pages	# of domains	# of initial errors
1	XXX is not defined	200	166	307
2	Cannot read property XXX of null	156	126	176
3	XXX is not a function	92	86	111
4	Unexpected token X	54	51	61
5	Cannot set property XXX of null	21	17	24
6	Invalid or unexpected token	18	12	21
7	Unexpected identifier	13	11	15
8	Script error for: XXX	8	3	10
9	The manifest specifies content that cannot be displayed on this browser / platform.	5	5	7
10	adsbygoogle.push() error: No slot	4	4	7
	53 different errors	555	466	826

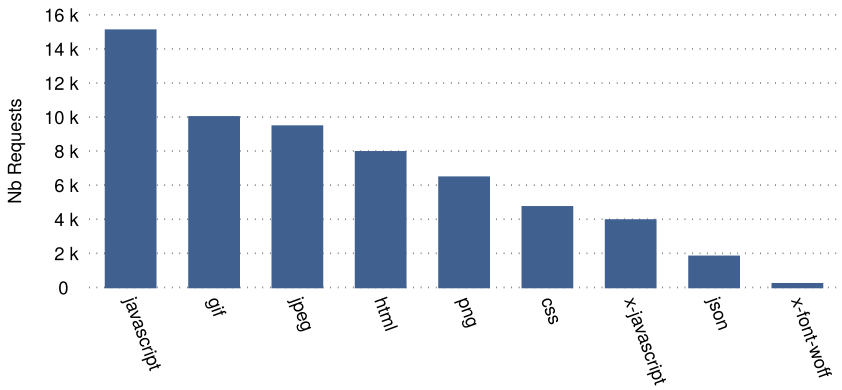


Figure 4. Bar plot of the number of requests by content-type.

large diversity of websites. There is, on average, 1.49 error per page, and each page has between 1 and 10 errors.

Table III presents the top 10 of the errors present in DeadClick. In total, DeadClick contains 53 different error types for a total of 826 collected errors; 69% of the JavaScript errors are the first three error types: `XXX is not defined`, `Cannot read property XXX of null` and `XXX is not a function`. Figure 4 presents the number of requests for the top 9 resource types. In our benchmark, the most common external resources are JavaScript files. The rest of the distribution illustrates how complex modern web pages are. For sake open of open-science, DeadClick and its mining framework are available on Github [10].

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table IV. The effectiveness of BikiniProxy and BugBlock (right-hand side).

#	Error messages	BikiniProxy		BugBlock	
		# of healed errors	Improvement	# of healed errors	Improvement
1	XXX is not defined	184	59.93%	36	11.72%
2	Cannot read property XXX of null	42	23.86%	10	5.74%
3	XXX is not a function	11	9.9%	20	18.01%
4	Unexpected token X	2	3.27%	8	13.11%
5	Cannot set property XXX of null	11	45.83%	0	0%
6	Invalid or unexpected token	0	0%	0	0%
7	Unexpected identifier	0	0%	0	0%
8	Script error for: XXX	2	20%	0	0%
9	The manifest specifies content that cannot be displayed on this browser/platform.	0	0%	0	0%
10	adsbygoogle.push() error:	0	0%	0	0%
	53 different errors	248/826	30.02%	88/826	10.67%

4.4. RQ1: Effectiveness of self-healing web applications

We now present the results of the first research question. Table IV shows the top 10 types of errors in the considered benchmark and how they are handled by BikiniProxy and BugBlock.

The first column contains the rank of the error type. The second column contains the error type, represented by the message of the error. The third column contains the number of healed errors with BikiniProxy. The fourth column contains the percentage of errors fixed with BikiniProxy. The fifth column contains the number of healed errors with BugBlock. The sixth column contains the percentage of errors fixed with BugBlock.

The first major result lies in the first row. It presents the error "XXX is not defined," which is the most common on the web according to our sampling. This error is present in 200 web pages across 166 different domains (Table III). It is thrown 307 times, meaning that some web pages throw it several times. With BikiniProxy, this error is healed 184/307 times, which represents a major improvement of 59.93%. With BugBlock, this error is healed 36/307 times, which represents a major improvement of 11.72%.

A second major result is that BikiniProxy is able to handle at least one error for the five most frequent JavaScript errors. It succeeds to heal between 3.27% and 59.93% of the five most frequent JavaScript errors in our benchmark. Overall, BikiniProxy handles 248 errors, and BugBlock handles 88 errors. It means that BikiniProxy reduces by 30.02% of the number of errors in the benchmark and BugBlock by 10.67%. Those results also indicate that the drawback of using a browser extension is almost 20% fewer handled error.

Now, we discuss the categories of healed errors. We identify whether

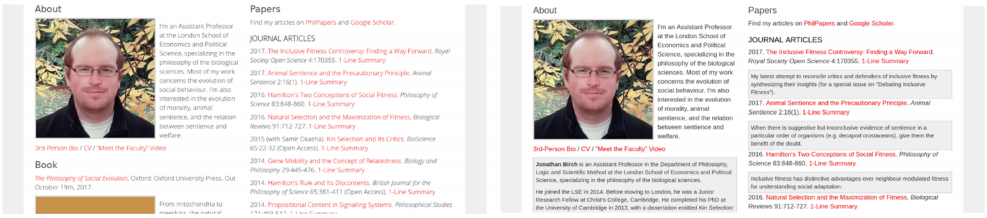
1. *All errors disappeared:* no error happens anymore in the page loaded with our tool, meaning that one or a combination of rewriting strategies have removed the errors.
2. *Some errors disappear:* there are fewer errors than in the original web trace.
3. *Different errors appear:* at least one error still, and it is a new error (new error type or new error location) that has never been seen before.
4. *No strategy applied:* the error type is not handled by any of the strategies, and thus, there are the same errors than in the original web trace.

Table V presents the number of web pages per category. The first line of Table V shows that the number of web pages that have all the JavaScript errors healed by BikiniProxy and BugBlock. BikiniProxy is able to handle all errors for 176/555 (31.76%) web pages of the DeadClick benchmark. BugBlock is able to handle all errors for 87/555 (15.67%) of the DeadClick benchmark. The second line shows the number of web pages that have been partially self-healed; by partially, we mean that the number of JavaScript errors decrease but are still not zero. With BikiniProxy, 42 web pages contain fewer errors than before, and with BugBlock, nine web pages are in this case. The third line shows the number of web pages that have new errors than before: with BikiniProxy, this

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Without BikiniProxy, some content is missing (B) Using BikiniProxy, the page loading is self-healed.

Figure 5. A real web page suffering from a JavaScript bug. With BikiniProxy, the bug is automatically healed, resulting in an additional information provided to the web page visitor.

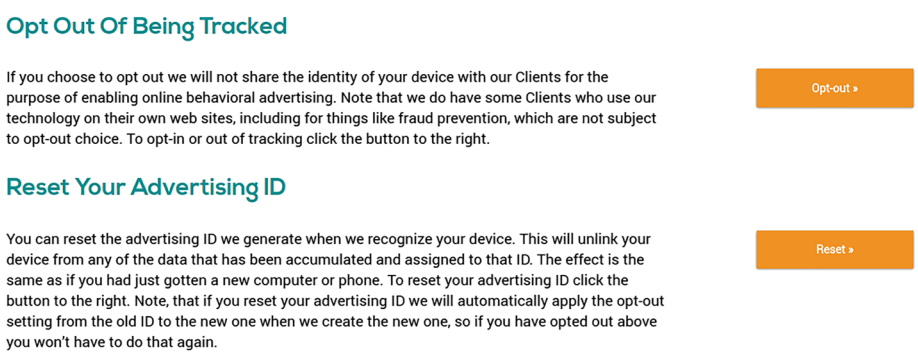

Opt Out Of Being Tracked
If you choose to opt out we will not share the identity of your device with our Clients for the purpose of enabling online behavioral advertising. Note that we do have some Clients who use our technology on their own web sites, including for things like fraud prevention, which are not subject to opt-out choice. To opt-in or out of tracking click the button to the right.
Opt-out »
Reset Your Advertising ID
You can reset the advertising ID we generate when we recognize your device. This will unlink your device from any of the data that has been accumulated and assigned to that ID. The effect is the same as if you had just gotten a new computer or phone. To reset your advertising ID click the button to the right. Note, that if you reset your advertising ID we will automatically apply the opt-out setting from the old ID to the new one when we create the new one, so if you have opted out above you won't have to do that again.
Reset »

Figure 6. The two buttons in orange are missing in the original buggy page. When BikiniProxy is enabled, the two orange buttons provide the user with new user interface features.

Table V. Analysis of the healing effectiveness per page.

Metric name	BikiniProxy		BugBlock	
	# of pages	Per cent	# of pages	Per cent
All errors disappeared	176/555	31.76	87/555	15.67
Some errors disappeared	42/555	7.58	9/555	1.62
Different/additional errors	140/555	25.27	52/555	9.37
No strategy applied	196/555	35.31	407/555	73.15

case is detected for 140/555 (25.27%) web pages and 52/555 (9.42%) with BugBlock. The last line shows the number of web pages where none of the strategies has been applied: in 196/555 (35.13%), the errors are of a type that is not considered by BikiniProxy. In the case of BugBlock, 406/555 (73.28%) of the buggy pages have not been handled by BugBlock. To better understand the case where no strategy can be applied, we perform a manual qualitative analysis.

4.4.1. Unhandled errors. The errors are unhandled when none of the five rewriting strategies succeed to heal the errors. In our experiment, this type of scenario is frequent for BugBlock with 73.28% of the web pages not being handled and in a lower proportion for BikiniProxy with 196/555 of web pages have errors not healed, which represents 35.13% of the erroneous web pages of DeadClick.

The difference between the number of unhandled errors between BikiniProxy and BugBlock is related to a technical limitation of BugBlock. BugBlock is not able to rewrite the HTML content of the page before they are executed. It means that in all the cases where the error is triggered inside an HTML page, for example, in a <SCRIPT> tag, BugBlock is not able to handle the error. This is due to a technical limitation of browser extensions.

For the other cases, we identify two main root causes. The first cause of non-healed errors is that the error type is not supported. For example, the web page http://dnd.wizards.com/articles/unearthed-arcana/artificer is loading a JSON file. However, the JSON file is invalid, and the browser

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

does not succeed to parse it, which produces an `Unexpected token <` error. None of the five strategies is able to handle malformed JSON errors. The second cause of non-healed errors is that the self-healing strategies have not enough information to rewrite the resource. For example, the web page http://moreas.blog.lemonde.fr/2007/02/28/le-pistolet-sig-sauer-est-il-adapte-a-la-police/ contains the error `Cannot read property 'parents' of undefined`, and this error should be healed with "Object Creator" rewriting. However, the trace of the error does not contain the URL of the resource that triggers this error because the JavaScript code has been unloaded. Consequently, "Object Creator" is not able to know which resource has to be rewritten to handle the error.

In summary, Table V shows that BikiniProxy is almost able to heal all the errors from a third of DeadClick. The second third of the benchmark is pages that cannot be healed with BikiniProxy. The last third contains web pages that are partially headed or that the self-healing strategies produce new errors. In the case of BugBlock, Table V shows that BugBlock is not able to handle the errors in the majority of the cases (73%) and is able to handle the errors of 15.67% of the pages completely. It shows that the technical limitations of browser extension have an important impact on the healing effectiveness of BugBlock.

Answer to RQ1. How are BikiniProxy and BugBlock effective at automatically fixing JavaScript errors in production, without any user or developer involvement? BikiniProxy is effective the handle the five most frequent JavaScript errors present in our benchmark. With the currently implemented self-healing rewriting strategies, BikiniProxy is able to fully heal 248/826 (30.02%) of all errors, representing 196/555 (31.76%) of all buggy web pages of our benchmark. The healing effectiveness of BugBlock is reduced by the technical limitation of browser extensions. Our experiment shows that 10.67% of the errors have been fully healed by BugBlock. This shows that despite being more practical, a browser extension has a lower performance.

4.5. RQ2: Outcome

In this second research question, we focus on category "All Errors Disappeared" and further refines the classification as follows:

1. The errors have disappeared, but the end-user can see no behavioural change.
2. The errors have disappeared, and new UI features (e.g. new buttons) are available to the end-user.
3. The errors have disappeared, and new content is available for the end-user.

Contrary to RQ1, it is not possible to automatically classify all pages with this refined category, because it requires a human-based assessment of what is new content or new features. For this reason, we answer this RQ with a qualitative case study analysis, and we do not consider BikiniProxy and BugBlock separately because the case studies are valid for both tools.

4.5.1. Error handled but no behaviour change. A healed error does not automatically result in a behaviour change in the application. For example, this is the case for the website https://cheapbotsdonequick.com/source/bethebot, which triggers the error `"module" is not defined`. This error is triggered by line `module.exports = tracery;`. This type of line is used to make a library usable by another file in a Node.js environment. However, Node.js has a different runtime from a browser, and the module object is not present, resulting in the error. With BikiniProxy, the self-healing strategy "Object Creator" automatically initializes the variable `module`; however, because this line is the last line of the executed JavaScript file, this has absolutely no further consequence on the execution or the page rendering. This means that the error was irrelevant. However, from a self-healing perspective, this cannot be known in advance. From a self-healing engineering perspective, the takeaway is that it is more straightforward to heal irrelevant errors than to try to predict their severity in advance.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.5.2. New feature available. One possible outcome of our approach is that the self-healing strategy unlocks new features. For example, this is the case of https://bluecava.com/. This page has an error, shown in Listing 1, which is triggered because the developer directly accesses the content of Ajax requests without checking the status of the request. However, there are requests that are denied due to cross-domain access restrictions implemented in all browsers. Because the developer did not verify if there is an error before accessing the property "id" on a null variable, the JavaScript event loop crashes.

With our approach, the self-healing strategy "Object Creator" ensures the initialization of the variable if it is null. This execution modification allows the execution to continue and to finally enter into an error handling block written by the developer, meaning that the event loop does not crash anymore. The execution of the page continues and results in two buttons being displayed and enabled for the end-user. Figure 6 presents the two buttons that are now available for the user.

Listing 1: Error on the web page https://bluecava.com/

```
Uncaught TypeError: Cannot read property 'id' of null
 at bluecava.js?v=1.6:284 ...
 at post (bluecava.js?v=1.6:40)
 at identify (bluecava.js?v=1.6:156) ...
```

4.5.3. New content available. One other outcome of our approach is that additional content is displayed to the end-user.

Let us consider the web page `http://personal.lse.ac.uk/birchj1/` that is the personal page of a researcher. This page triggers the following error: `$ is not defined at (index):20`

This error is thrown because a script in the HTML page calls the jQuery library before the library is loaded. The script that throws the error is responsible for changing the visibility of some content on the page. Consequently, because of the error, this content stays hidden for all visitors of the page.

Using our tool, the error is detected as being caused by a missing jQuery library. This error is healed by rewriting strategy "Library Injector." Consequently, the missing jQuery library is injected in the buggy page. When the rewritten web page is executed, jQuery is available, and consequently, the script is able to change the visibility of hidden HTML elements, resulting in a newly visible content.

Figure 5 presents the visual difference between the original page (left side) and loaded with BikiniProxy (right side). All the elements in a grey box on the right-hand side are missing on the left image. They have appeared thanks to self-healing.

Finally, we have manually checked the presence of potentially harmful effects. By manually analysing a random sample of 25 self-healed subjects, we did not find a single harmful effect.

Answer to RQ2. What is the outcome of self-healing strategies with BikiniProxy and BugBlock on the page beyond making the error disappear? We observe three outcomes in our benchmark: (i) no visible change, (ii) new features and (iii) new content. BikiniProxy and BugBlock are able to restore broken features or broken content automatically. We have not observed any harmful effect of speculative execution.

4.6. RQ3: Strategies

In this research question, we compare the five different self-healing strategies. For each strategy, Table VI shows the number of times it has been activated to heal errors of DeadClick, with our two tools: BikiniProxy and BugBlock. The last column presents the number of different error types

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table VI. The number of activations of each self-healing strategy and the number of error types that the strategy can handle.

Self-healing strategies	# of activations		# of supported error types
	BikiniProxy	BugBlock	
Line Skipper	233	89	4
Object Creator	109	17	2
Library Injector	75	55	3
HTTP/HTTPS Redirector	18	18	NA
HTML Element Creator	14	11	2

for which the strategy has been selected. For example, the first row of Table VI shows that "Line Skipper" has been selected to handle 233 errors with BikiniProxy and 89 with BugBlock, and it has healed four different error types.

In the case of BikiniProxy, the most used strategy is "Line Skipper" with 233 activations. It is also the strategy that supports the highest number of different error type: (i) "XXX is not defined," (ii) "XXX is not a function," (iii) "Cannot read property XXX of null," and (iv) "Cannot set property XXX of null." On the other hand, BugBlock uses most the strategy "Line Skipper" with 89 activations, followed by "Library Injector" with 55 activations. The second most used strategy for BikiniProxy is "Object Creator" with 109 errors for which it has initialized a null variable. This strategy handles two different error types: "Cannot set property XXX of null" and "Cannot read property XXX of null." These two strategies have something in common; they target the failure point, the symptom, and not the root cause (the root cause is actually unknown). For example, the error `CitedRefBlocks is not defined` is triggered in the web page https://www.ncbi.nlm.nih.gov/pmc/articles/PMC504719/, because the function `CitedRefBlocks` is not defined. Line Skipper strategy avoids the error by skipping the method call; it is a typical example of a fix at the failure point and not at the root cause of the absence of `CitedRefBlocks`.

On the contrary, "Library Injector" addresses the root cause of the problem: the missing library is extracted from the error message, and it is used to rewrite the content of the request. In this case, the self-healing tools exploit the fact that we have a direct relation between root cause (no included library) and symptom (unknown used library name) for this error type.

The case of "HTTP/HTTPS Redirector" is the opposite. Recall that "HTTP/HTTPS Redirector" directly looks in the HTML body of the resource if there are scripts that will be blocked. This means that the rewriting addresses the root cause of potential future problems. For example, the page https://corporate.parrot.com/en/documents tries to load the resource http://www.google-analytics.com/urchin.js, but the request is blocked by the browser (HTTP request in an HTTPS page). Consequently, the Google tracking library is not loaded, and function `urchinTracker` is not defined, resulting in the error `urchinTracker is not defined`. "HTTP/HTTPS Redirector" strategy rewrites the URL of the resource in the <SCRIPT> tag to https://www.google-analytics.com/urchin.js, and this fixes the error of the page. This strategy can potentially fix error types that we cannot envision. Hence, we do not know the exact number of handled error types, so we put "NA" in Table VI. Finally, strategy "HTML Element Creator" is applied to more rare errors happening only 14 times in our benchmark with BikiniProxy and 11 times with BugBlock.

In this research question, we also observe that the number of activations is lower for BugBlock. This observation is directly related to our observation of unhandled errors in the research question one (subsubsection 4.4.1).

5. DISCUSSION

5.1. Security analysis

BikiniProxy and BugBlock are founded on the core failure-oblivious computing principle [5]: any execution happening after the avoided failure is, in essence, speculative. This speculative execution must be sandboxed.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Answer to RQ3. Do the different self-healing strategies perform equivalently? In our experiment, the most used strategy is "Line Skipper" for BikiniProxy because it is able to heal from four common error types with the same strategy. This strategy is not the most frequent in the case of BugBlock because it is not able to modify the HTML page of an application. Other self-healing strategies can be designed and added to BikiniProxy and BugBlock in order to address the rare error types in the long tail of field errors.

The security guarantees of BikiniProxy and BugBlock are provided by the sandboxing in the browser and on the server-side. First, all browsers contain very carefully engineered code to sandbox the execution of JavaScript code. This sandboxing means that (i) the JavaScript code cannot access or transfer data to other tabs and windows (aka tab sandboxing), (ii) the JavaScript code cannot access or transfer data to other websites (cross-domain restrictions) and (iii) the JavaScript code cannot access to the file system.

Second, in distributed Internet applications with code running on the server-side and on the client-side, it is known that one cannot trust the execution of the client code. Consequently, the best practice is to protect the server-side state with appropriate checks in the REST API accessed by client-side JavaScript. Those checks form the second sandboxing of speculative execution of this approach: the unwanted side-effects are confined to the current browser window.

In terms of privacy, the only information that is shared between BikiniProxy, BugBlock and the backend is the stacktraces of errors. The stacktraces do not contain personal information or information that can lead to identifying a specific user or its browsing habit. With BugBlock, this can even be addressed: BugBlock could be extended by including the backend service directly inside the browser extension. Using this approach, no information ever is sent by BugBlock to a third party server.

5.2. Applicability analysis

The usage of BikiniProxy and BugBlock is practically zero cost, and as such, it is widely applicable. First, it requires no change to the original web pages or applications. Second, for the BikiniProxy case, the usage of an HTTP proxy in web applications is very common. A BikiniProxy self-healing proxy can be set up by (i) a company in front of their web content, (ii) a SaaS-based provider and (iii) a hosting service. BikiniProxy targets the professional sector, while BugBlock targets the end-user that only needs to click on one button to install the extension in its browser. Thus, BikiniProxy and BugBlock target two different public.

6. THREATS TO VALIDITY

We now discuss the threats to the validity of our experiment. First, let us discuss internal validity. Our experiment is relying on the implementation of our prototype; consequently, a bug in our code may threaten the validity of our results. However, because the source code of the approach and of the benchmark is publicly available [10], future researchers will be able to identify these potential bugs. It is unknown whether the errors of our benchmark are representative of all errors in the web and whether 96 174 visited pages is enough compared with the trillions of pages of the Internet. To our knowledge, there is no work on the representativeness of JavaScript bugs.

Our approach has been carefully designed to maximize representativeness: (i) the randomness of keyword choice allows us to discover websites about many different topics, carried out by a variety of persons, with different backgrounds (a website on CSS performed by a web developer is likely to have fewer errors than a website on banana culture performed by a hobbyist) and (ii) the ranking of Google for a specific query provides us with a filter that favours popular websites. If errors are detected on those websites, they likely affect many users.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

7. RELATED WORK

7.1. JavaScript error and repair

Several studies on client-side JavaScript have been made by Ocariza et al [2, 12, 24]. In 2011, they showed that most websites contain JavaScript errors even in the top 100 of Alexa [2]. They have also investigated [12, 24] the nature of the JavaScript errors as follows. They manually analyse JavaScript bug reports from various web applications and JavaScript libraries. They find that the majority of reported JavaScript bugs are related to the DOM. None of this work has explored self-healing strategies for the web.

Now, we discuss the works on reproducing JavaScript errors or to extract regression tests. Wang et al. [25] present a technique to reproduce sequences of events that lead to a JavaScript error. Schur et al. [26] present a fully automatic tool to generate test scripts based on the behaviour of multi-user web applications. The goal of those works are different from ours: they focus on creating tests while we focus on healing the error on the fly, in production.

Hanam et al. [27] present BugAID, a data mining technique for discovering common unknown bug patterns in server-side JavaScript. Hanam et al. focus on server-side bugs; on the contrary, BikiniProxy targets client-side JavaScript code, and it heals the errors on the fly.

There have been several repair tools targeting JavaScript front-end code. Ocariza et al. [28] present Vejovis, a technique that suggests JavaScript code modifications to handle DOM-related errors. Pradel et al. [29] and Bae et al. [30] proposed tools for detecting type inconsistencies and web API misuses in JavaScript, respectively. They also present common fault types and common web API misuse patterns. Roy et al. [31] present X-PERT, an automatic technique to detect cross-browser issues in web applications. Those works are offline program repair requiring test cases [32], while BikiniProxy is an online self-healing, in production, without the developer in the loop.

7.2. Self-healing in production

There are many kinds of self-healing approaches, and we refer the reader to recent surveys on this topic [33, 34]. We now concentrate on self-healing for the web.

Carzaniga et al. [8] propose a technique that automatically applies workarounds to handle API issues. The workarounds are based on a set of manually written API-specific alternative rule. The difference with our work is that we defined five self-healing strategies that are generic, that is, which are not application specific. On the contrary, the work of Carzaniga et al. relies on specific templates for specific APIs, and new templates have to be manually created to support new APIs. In subsequent work, the same group has proposed a way to automatically mine those workarounds [9]. However, those workarounds do not consider generic JavaScript errors as we do; they target API specific errors for which workarounds have been identified or mined.

Several approaches also use a proxy-based architecture in their contribution. Kiciman et al. [35] present a web proxy named AjaxScope, which instruments the JavaScript code to monitor the performance of web applications. It does monitoring and not self-healing. Zhang et al. [36] present a technique to change the user interface of mobile applications on the fly. In particular, they aim at providing accessible UIs to blind users. Appelt et al. [37] do automatic repairs of firewall rules to improve the security of web application. While a firewall and a proxy are similar, the goal and the means are different: they focus on security while we focus on availability; they change firewall rules while BikiniProxy rewrites HTML and JavaScript code.

8. CONCLUSION

In this paper, we have presented a novel approach to provide self-heal capabilities for the web, focusing on client-side JavaScript errors and two different implementations of this approach: an HTTP proxy called BikiniProxy and a browser extension called BugBlock. We have evaluated our technique on 555 web pages with JavaScript errors, randomly collected on the web. Our qualitative and quantitative evaluation has shown that BikiniProxy is effective, and self-healing results in providing the web user with new features and content. It also shown that BugBlock is able to heal buggy

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

web pages but with a smaller proportion due to technical limitations of browser extensions. Future work is required to devise new self-healing rewriting strategies for solving the maximum number of JavaScript runtime errors.

REFERENCES

1. Surveys W3Techs World Wide Web Technology. *Usage ranking of JavaScript libraries*, 2018. https://w3techs.com/technologies/overview/Javascript_library/all.
2. Ocariza Jr FS., Pattabiraman K, Zorn B. JavaScript errors in the wild: an empirical study. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on IEEE*, 2011.
3. Keromytis AD. Characterizing self-healing software systems. *Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, 2007; 22–33.
4. Koopman P. Elements of the self-healing system problem space. *Proc. Workshop on Architecting Dependable Systems*, 2003.
5. Rinard MC., Cadar C, Dumitran D, Roy DM., Leu T, Beebee WS. Enhancing server availability and security through failure-oblivious computing. In *Osdi*, Vol. 4, 2004.
6. Durieux T, Cornu B, Seinturier L, Monperrus M. Dynamic patch generation for null pointer exceptions using metaprogramming. *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on IEEE*, 2017; 349–358.
7. Candea G, Kiciman E, Zhang S, Keyani P, Fox A. Jagr: An autonomous self-recovering application server. *Autonomic Computing Workshop. 2003. Proceedings of the IEEE*, 2003; 168–177.
8. Carzaniga A, Gorla A, Perino N, Pezzè M. Automatic workarounds for web applications. *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*: ACM, 2010; 237–246.
9. Carzaniga A, Gorla A, Perino N, Pezze M. Automatic workarounds: exploiting the intrinsic redundancy of web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2015; **24**(3):16.
10. Durieux T, Hamadi Y, Monperrus M. *Bikiniproxy repository*, 2018. https://github.com/Spirals-Team/bikiniproxy/.
11. Durieux T, Hamadi Y, Monperrus M. Fully automated HTML and JavaScript rewriting for constructing a self-healing web proxy. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018; 1–12.
12. Ocariza FS., Bajaj K, Pattabiraman K, Mesbah A. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering* 2017; **43**(2):128–144.
13. Luotonen A. *Web proxy servers*. Prentice-Hall, Inc., 1998.
14. Pistriotto JC., Montinola K. *Method and apparatus for configuring a client to redirect requests to a caching proxy server based on a category ID with the request*. Google Patents, 2000. US Patent 6,138,162.
15. Krueger T, Gehl C, Rieck K, Laskov P. Tokdoc: a self-healing web application firewall. *Proceedings of the 2010 ACM Symposium on Applied Computing*: ACM, 2010; 1846–1853.
16. Bowman-Amuah MK. *Load balancer in environment services patterns*. Google Patents, 2003. US Patent 6,578,068.
17. Gorla A, Pezzè M., Wuttke J, Mariani L, Pastore F. Achieving cost-effective software reliability through self-healing. *Computing and Informatics* 2010; **29**:93–115.
18. Berger E. Software needs seatbelts and airbags. *ACM Queue* 2012; **10**.
19. Lewis C, Whitehead J. Runtime repair of software faults using event-driven monitoring. *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, Vol. 2: IEEE, 2010; 275–280.
20. Shehory O, Martinez J, Andrzejak A, Cappiello C, Funika W, Kondo D, Mariani L, Satzger B, Schmid M. Self-healing and recovery methods and their classification. In *Self-healing and Self-adaptive Systems*, Andrzejak Artur, Geihs Kurt, Shehory Onn, Wilkes John (eds), Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany: Dagstuhl, Germany, 2009.
21. Gu T, Sun C, Ma X, Lü J., Su Z. Automatic runtime recovery via error handler synthesis. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*: ACM, 2016; 684–695.
22. Burton S. *We crunched 1 billion java logged errors - hereâĂŹs what causes 97% of them*, 2018. https://blog.takipi.com/we-crunched-1-billion-java-logged-errors-heres-what-causes-97-of-them/.
23. Callegati F, Cerroni W, Ramilli M. Man-in-the-middle attack to the https protocol. *IEEE Security & Privacy* 2009; **7**(1):78–81.
24. Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side JavaScript bugs. *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*: IEEE, 2013; 55–64.
25. Wang J, Dou W, Gao C, Wei J. JSTrace: fast reproducing web application errors. *Journal of Systems and Software* 2017.
26. Schur M, Roth A, Zeller A. Procrawl: Mining test models from multi-user web applications. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*: ACM, 2014; 413–416.
27. Hanam Q, Brito Fernando S de M, Mesbah A. Discovering bug patterns in JavaScript. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*: ACM, 2016; 144–156.
28. Ocariza Jr FS., Pattabiraman K, Mesbah A. Vejovis: suggesting fixes for javascript faults. *Proceedings of the 36th International Conference on Software Engineering*: ACM, 2014; 837–847.
29. Pradel M, Schuh P, Sen K. Typedevil: Dynamic type inconsistency analysis for JavaScript. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*: IEEE Press, 2015; 314–324.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

30. Bae S, Cho H, Lim I, Ryu S. Safewapi: web API misuse detector for web applications. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*: ACM, 2014; 507–517.
31. Roy Choudhary S, Prasad MR., Orso A. X-pert: accurate identification of cross-browser issues in web applications. *Proceedings of the 2013 International Conference on Software Engineering*: IEEE Press, 2013; 702–711.
32. Kong X, Zhang L, Wong WEric, Li B. Experience report: how do techniques, programs, and tests impact automated program repair?. *26th IEEE International Symposium on Software Reliability Engineering*, 2015; 194–204.
33. Monperrus M. Automatic software repair: a bibliography. *ACM Computing Surveys* 2017; **51**:1–24.
34. Gazzola L, Micucci D, Mariani L. Automatic software repair: a survey. *IEEE Transactions on Software Engineering* 2018.
35. Kiciman E, Livshits B. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, ACM: New York, NY, USA, 2007; 17–30. http://doi.acm.org/10.1145/1294261.1294264.
36. Zhang X, Ross AS., Caspi A, Fogarty J, Wobbrock JO. Interaction proxies for runtime repair and enhancement of mobile application accessibility. *Proceedings of the 2017 Chi Conference on Human Factors in Computing Systems*: ACM, 2017; 6024–6037.
37. Appelt D, Panichella A, Briand LC. Automatically repairing web application firewalls based on successful SQL injection attacks. *28th IEEE International Symposium on Software Reliability Engineering*, 2017; 339–350.

© 2020 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

10991689, 2020, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1731 by Kungliga Tekniska Högskolan, Wiley Online Library on [28/08/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License